Job preemption with PBS

Some Derecho jobs are suitable for running in the system's "preempt" queue on an as-available basis on resources that would otherwise be idle. Running them in that queue allows them to be preempted with minimal impact when a higher-priority job requires the use of those resources. Suitable workflows include those with short or fairly unpredictable runtimes; for example, data processing, file movement, or running analysis tools that have an efficient *checkpoint/restart* capability.

Page contents

Using the preempt queue

The "preempt" queue is similar to the "main" Derecho queue in that it serves to route jobs to the system's CPU or GPU nodes. It is different, however, in that:

- Jobs will start only when they can make use of resources that would otherwise be idle.
- They will be terminated – with a 10-minute grace period – if the resources being used are required by a higher priority job in a different queue.

The start time of a job in the preempt queue will be unpredictable because of the idle resource requirement. Once it starts, it is guaranteed at least 10 minutes of runtime but potentially much more. The duration depends on jobs other users submit to PBS after your job begins.

To submit a job to the preempt queue, simply specify preempt as the queue name in your PBS script header as follows:

```
#!/bin/bash
#PBS -A project_code
#PBS -N preemptable_job
#PBS -q preempt
#PBS -r n
#PBS -l walltime=04:00:00
#PBS -l select=2:ncpus=128:mpiprocs=32:ompthreads=4

...
```

The **walltime** specification is the job duration upper limit. Use the specifier **#PBS -r** to indicate if the job should be rerun if it is preempted; valid options are **y** or **n** for yes or no. All other aspects of the PBS script are unchanged by the use of preemption.

Abrupt termination may be entirely acceptable for some workflows. This could be the case for batch removal of a large number of files, for example, or if the application writes frequent checkpoint files and can restart successfully after being interrupted. In other cases, it may be beneficial for the application to take a specific action within the 10-minute grace window. Such an approach is possible with minor changes to the application as described below.

Practical implications for *preempt* throughput

Idle resources are a prerequisite for jobs in the preempt queue to start. The smaller the resource request, the more likely there will be an idle portion of the machine on which to schedule the job. Conversely, large jobs in the queue are likely to wait for long periods of time, if they execute at all. The ideal use case is small-to-medium sized jobs that are robust to interruption and that can make meaningful progress in short periods of time.

Charging and allocations

Jobs run in the preempt queue are charged at a queue factor of only 0.2, less than jobs in the "economy" queue. Jobs that do not run to completion because of preemption are not charged against your allocation.

Terminating proactively with signal handling

When a job running in the preempt queue is targeted for preemption, PBS notifies the running process through a UNIX signal. PBS then waits 10 minutes before killing the process. A properly configured application can receive the notification signal, act upon it (typically through an existing checkpoint functionality), and then terminate gracefully rather than be terminated abruptly at the end of the grace period. The steps required to configure an application in this manner are:

1. Provide a signal handler function to receive the termination request.

2. Register the signal handler function with the operating system.
3. Invoke or add checkpoints to be triggered by the signal handler.

Steps 1 and 2 are fairly common across applications and even programming languages. Step 3 is application-specific, and usually involves writing the application state to disk so that it can be restarted later. For some applications, however, an even simpler approach may be possible. For example, if the target application is a data-processing pipeline, it may suffice to receive the termination notification, complete the current processing step, and simply exit without beginning additional steps in the pipeline.

Signal handling and registration

C/C++ and Fortran

For traditional compiled languages such as C/C++ and Fortran, signal handling is most readily accomplished through some minimal C functions, even inside a predominantly Fortran application. This is because the operating system application interface is C based. The following shows the minimal required steps.

Sample C program to catch signals sent from the operating system

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
static int checkpoint_requested = 0;

void my_sig_handler (int signum)
{
  time_t now;
  time(&now);

  switch (signum)
    {
    case SIGINT:
    case SIGTERM:
    case SIGUSR1:
      checkpoint_requested = 1;
      printf("...caught signal %d at %s", signum, ctime(&now));
      break;

    default:
      printf("...caught other unknown signal: %d at %s", signum, ctime(&now));
      printf("   see \"man 7 signal\" for a list of known signals\n");
      break;
    }
}

int main (int argc, char **argv)
{
  /* register our user-defined signal handlers */
  signal(SIGINT,  my_sig_handler);
  signal(SIGTERM, my_sig_handler);
  signal(SIGUSR1, my_sig_handler);

  return 0;
}
```

First, we declare a C function **my_sig_handler**, which takes the signal identifier as input. In this example we construct a switch statement that allows for processing different types of signals in the same code block. It is evident from the listing that if the function is called with a **SIGINT**, **SIGTERM**, or **SIGUSR1** signal then we set the flag **checkpoint_requested** and print an informative statement. For completeness, if called with any other signal, we print a diagnostic message as well but take no other action.

Second, we call the system routine **signal()** to register our function for the specific signals we want the application to process. In this case, we are asking the operating system to call our function my_sig_handler() any time a **SIGINT**, **SIGTERM**, or **SIGUSR1** is encountered.

The third step is application specific and not listed, but the general idea is elsewhere in the application (for example, the main time step loop) we would check the value of the **checkpoint_requested** flag and take appropriate action to save state and exit gracefully.

See a complete MPI/C++ example here.

To integrate such an approach into a Fortran application, it is simplest to create a C function taking no arguments that encapsulates the signal registration process and calling it from within your Fortran main application. Please contact CISL help for further assistance.

While the most common use case is compiled languages as shown above, it is also possible to catch and act upon signals when your main application is a shell script or in Python, as shown below.

Shell scripts

In shell scripting, the process is generally similar, with some very slight changes in terminology. Notably, in shell scripts 'traps' can be used to intercept signals and call a user-specified function, in much the same way a signal handler can be installed in a compiled language. A complete example follows.

Sample bash script to catch signals sent from the operating system

```
#!/usr/bin/env bash

checkpoint_requested=false

function my_signal_handler()
{
    case ${sig} in
        SIGINT)
        SIGTERM)
        SIGUSR1)
            printf "...caught %s at %s\n" ${sig} "$(date)"
            checkpoint_requested=true
            ;;
        *)
            printf "...caught unknown signal: %s\n" ${sig}
            exit 1
            ;;
    esac
}

function register_signal_handler ()
{
    printf "Registering user-specified signal handlers for PID %d\n" $$

    trap "sig=SIGINT;  my_signal_handler" SIGINT
    trap "sig=SIGTERM; my_signal_handler" SIGTERM
    trap "sig=USR1;    my_signal_handler" SIGUSR1
}

function do_checkpoint ()
{
    for i in $(seq 1 10); do
        printf "\t%2d : Inside checkpoint function\n" $i
        sleep 5s
    done

    checkpoint_requested=false;
}

# main function follows
register_signal_handler

for i in $(seq 1 50); do
    printf "%2d : Main function\n" $i
    sleep 5s

    if ${checkpoint_requested}; then
        do_checkpoint
    fi
done
```

Running the previous code will enter a "Main Function" loop that executes a number of steps. Sending **Control+C** to the running program effectively sends a SIGINT and invokes the desired signal handling function through the bash trap mechanism.

Sample python script to catch signals sent from the operating system

Finally, Python provides the **signal** module, which can be used in a user application to catch signals from the operating system as shown here:

```
#!/usr/bin/env python3

import signal
import sys
import time
import os

from datetime import datetime

checkpoint_requested = False

def my_signal_handler(sig, frame):
    global checkpoint_requested

    if signal.SIGINT == sig or signal.SIGTERM == sig or signal.SIGUSR1 == sig:
        print("...caught signal {} at ".format(sig) + datetime.now().strftime("%H:%M:%S %B %d, %Y"))
        checkpoint_requested = True
    else:
        print("...caught unknown signal: {}".format(sig))
        sys.exit(1)
    return

def register_signal_handler ():
    print("Registering user-specified signal handlers for PID {}".format(os.getpid()))
    signal.signal(signal.SIGINT,  my_signal_handler)
    signal.signal(signal.SIGTERM, my_signal_handler)
    signal.signal(signal.SIGUSR1, my_signal_handler)
    return

def do_checkpoint ():
    global checkpoint_requested

    for i in range(1, 11):
        print("\t{:2d} : Inside checkpoint function".format(i))
        sys.stdout.flush()
        time.sleep(5)

    checkpoint_requested = False;
    return

if __name__ == "__main__":
    register_signal_handler();

    for i in range(1, 51):
        print("{:2d} : Main function".format(i))
        sys.stdout.flush()
        time.sleep(5)

        if checkpoint_requested:
            do_checkpoint()
```

Additional resources

All the sample scripts above are available through the NCAR hpc-demos GitHub repository in the PBS/preempt subdirectory.

- More detail on signal handling in C
- More detail on using traps in shell scripts
- More detail on signal handling in Python